Unit Testing In Java With JUnit

Module 1: JUnit Overview
- What JUnit is and unit testing
- Setting up JUnit

Module 2: JUnit Basics
- The stuff you NEED to know

Module 3: Advanced JUnit
- Optional features that can be very useful

Module 4: Integration JUnit
- Working with build tools
- Reporting results, etc.

Module 5: Beyond JUnit
- Complementary tools
- Other uses besides unit testing

```
import static org.junit.Assert.*;

public class TrackingServiceTest {

    @Test
    public void test(){
        assertEquals(5, "Hello".length());
    }
}
```

Asserts
- Allows you to specify, what you expect and compare it against what you actually got.

Test setup and teardown
- Setup test data before it runs and tear it down after (makes sure our tests aren't repetative.

Exceptions Testing
- Used to verify if an exception was or wasn't thrown.

Test Suites
- Useful for managing a large amount of tests

Parameterized Testing
- Create tests that operate on sets of data that you feed into those tests.

Assumptions
- Used to ignore tests that aren't qualified to run on a particular platform or context

Rules and Theories
- Rules allow us to extend the functionality of JUnit by adding behaviours to tests whenever we run a particular rule.
- Theories are special types of rules that run under different kinds of tests.

Integration with popular build systems
- Ant + Maven

JUnit Methods and Annotations

```
public void <name of test>() {
}
```

*Test name should be as descriptive as possible.

- @Test

- @Before / @After - Allows you to specify a particular behaviour before and after each test is run.
- @BeforeClass / @AfterClass - Allows you to specify behaviour before any methods are run and after all methods are run.
- @Ignore - Tells the JUnit runner to ignore this test, usually used for disabling tests.
  ⇒ If you've got a test you don't want to run all the time.
- @Test (expected = Exception.class) => Allows us to expect that an exception will be thrown in our test.
- @Test (timeout = 100) => Specify a period our test should be executed in or it will timeout.

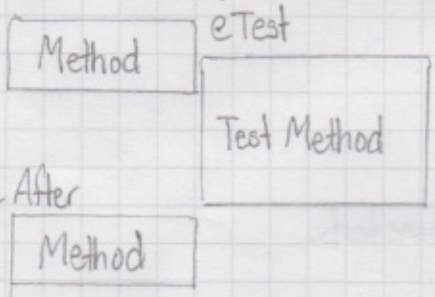
```
public class TrackingServiceTest {
    @Test
    public void NewTrackingServiceTotalIsZero()
    {
        TrackingService service = new TrackingService();
        assertEquals("Tracking service total was not zero", 0, service.getTotal());
    }
```

0 = expected, service.getTotal() = actual

```
    @Test
    public void WhenAddingProteinTotalIncreasesByThatAmount(){
        TrackingService service = new TrackingService();
        service.addProtein(10);
        assertEquals(10, service.getTotal());
    }
}
```

=> The @Before and @After annotations are useful for eliminating duplication in unit tests.

@Before (Usually given a name like set up).



JUnit + Software Testing

Classes that implement interfaces, should follow the convention of having the same name as the interface w. Impl suffix unless instructed otherwise.

For example, one class implements ContactManager => ContactManagerImpl.

Be careful with this, as it may interfare w. automatic tools that will analyse your code

fig1
```
@Test(timeout = 1000)
public void testsThatFinishedBeforeOneSecond(){
    //...
}
```

fig2
```
@Test(expected = IndexOutOfBoundsException.class)
public void testsNegativeIndecesFail(){
    //...
}
```

If the method doesn't throw IndexOutOfBoundsException in the test, the test will fail.

Initialisation: Before and After

```
@Before
public void buildUp(){
    // A file is created here to be used in every test:
@After
public void cleanUp(){
    // the file is deleted here, after each test ends
}
```

ie: Assuming that your testing class, contains three testing methods, the execution path would be: buildUp, first test, cleanUp, buildUp, second test, cleanUp, buildUp, third test, cleanUp.

Mock Objects = Create an object that imitates the behaviour of

```
public class PatientMock implements Patient {
    public String getName(){
        return "Mock"
    }
    public int getRelativeCount(){
        return 5;
    };
    // ... implementations of other methods here...
}
```

Test Driven Development

This is a programming methadology that tests should be driven before the actual program.

The TDD methadology is usually described as consisting of three steps that are repeated in a loop:

1. Write the tests for the next functionality / feature of the program.
2. Write the minimal code that passes all the new tests
3. Refactor the code to make it clearer and simpler. Run the tests at the end to make sure the final functionality is right.

For the sake of clarity we are going to describe this cycle with a few additional details (optional steps do not take place on every iteration of the loop, only some times):

1. Optional. Create or update the interface for the class that is going to be updated (be it fixing an error or adding a feature).
2. Create the new test
3. Optional. Create any needed structure to make the code compile: eg create the class and/or the method to test it if it did not exist.
   => No real code yet, just enough to compile: return null for any complex type, 0 for ints, etc.
4. Run the test and verify that it fails. If it does not fail, that usually means (a) it is not testing anything that was not tested before (redundant), or (b) it is incorrect (the test, not the unimplemented feature) and should be fixed, or (c) you wrote too much code in the previous step. In some rare cases, none of this will be true and the test will still pass.
   eg: when testing that the length of an empty list is 0.
5. Write the minimum code necessary to make the test pass. Do not write anything beyond that. If you think some important functionality is missing, write a test for it in the next iteration of the TDD loop and complete the implementation of the method.
6. Refactor the code if needed for simplicity + clarity
7. Start the cycle again with a new feature.

Four Main Benefits To This Style of Programming:

- As the tests are written in advance, all the code is tested by at least one testing method. Otherwise, programmers can forget to test some methods or be lazy about it (and often will).
- Errors are detected early, when they are cheap to fix. It is more difficult for errors to remain undetected until later in the development, when it can be more costly to fix them.
- Writing the tests first makes the programmers think about the real specification of the class or method, focusing on what needs to be done before their short-term memory is filled with how to do it.

Think

Unit Testing In Java With JUnit
Assertions

- assertArrayEquals
- assertEquals => If object = null, but you might also use assertNull, assertNotNull
- assertTrue > boolean methods
- assertFalse
- assertNull
- assertNotNull
- assertSame > Same or not the same exact objects.
- assertNotSame
- fail => Use when you want to automatically fail the tests.

Advanced
07_Creating Parameterized Tests